TP1

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Sebastián Fernandez Ledesma	392/06	sfernandezledesma@gmail.com
Fernando Gasperi Jabalera	56/09	fgasperijabalera@gmail.com
Maximiliano Wortman	892/10	maxifwortman@gmail.com
Santiago Camacho	110/09	santicamacho90@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Problema 1: Puentes sobre lava caliente

1.1. Presentación del problema

Se quiere atravesar un puente con n tablones dando saltos acotados por un valor de x tablones. Se empieza afuera del puente y se pretende salir completamente de éste, es decir que como mínimo hay que saltar una vez (en el caso trivial de que $x > n$). La dificultad consiste en que ciertos tablones conocidos están rotos, y no pueden ser pisados. Lo que pide el problema es minimizar la cantidad de saltos para atravesar el puente, o aclarar que es imposible. Los puentes estarán definidos como t_1 t_2 ... t_n donde $t_i = 0$ si el tablón está sano o $t_i = 1$ si está dañado.
Por ejemplo, podríamos tener el puente 0 1 0 0 con un salto máximo igual a 2. Como se arranca afuera, saltar al primer tablón se considera como un salto de 1 tablón. En este caso no podemos saltar los dos tablones permitidos porque el segundo tablón está roto (el puente, usando X para marcar donde estamos parados, se vería así: X 1 0 0). El segundo salto sí podremos saltar los 2 tablones, quedando 0 1 X 0, y con el tercer salto saldremos del puente.
Una configuración más complicada podría ser el puente 0 0 1 0 0 0 1 1 0 0 para un salto máximo de 3 tablones, ya que ahora tenemos dos posibilidades: saltar al primer o al segundo tablón. Usaremos un algoritmo goloso para resolver el problema (saltar la mayor cantidad posible de tablones) y demostraremos que es correcto y que es la solución óptima para el problema.

1.2. Resolución

1.2.1. Algoritmo

Dado este problema de optimización planteamos resolverlo con un algoritmo goloso, que consiste en seguir "una heurística consistente para elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima" [Cormen p.414 (Greedy Algorithms)]. El problema a optimizar es encontrar la mínima cantidad de saltos para cruzar el puente, y la decisión golosa o la opcion óptima en cada paso local es elegir el tablon más lejano que pueda alcanzar el participante de acuerdo al rango de salto que tenga.

El algoritmo recibe un vector con los tablones del puente (puente[i]) y un entero que representa el máximo salto que puede dar el participante (*maxSalto*).

Teniendo esa información inicializamos la variable *actual* y *proximo* en 0, que son enteros. La primera representa en que posición del puente se ubica el participante y la segunda la posición del salto más lejano que puede alcanzar a un tablon. Estas variables son actualizadas por un ciclo, que en el caso que haya solución corre hasta que la posición *actual* sea mayor a la cantidad de tablones, es decir que el participante haya cruzado el puente.

Dentro del ciclo, se calcula la variable *proximo* con una función (*calcularProximoTablon*) que recibe el *puente* la posición *actual* y el *maxSalto* y prueba desde el salto más largo que puede dar hasta el mínimo cual es el próximo tablon óptimo, si no existe, entonces devuelve una excepción y hace que el algoritmo termine o en caso contrario el ciclo lo guarda en un vector de *saltos*. *Actual* se actualiza a la posición *proximo* en cada iteración que significa que el participante avanza en cada vuelta del ciclo.

Una vez que termina el ciclo el algoritmo devuelve el arreglo de *saltos*, que es vacio si no existe solución.

1.2.2. Pseudocódigo

Algorithm 1 cruzarPuente(vector<int> puente, int maxSalto) → vector<int> saltos

```
int cantidadTablones ← |puente| − 2 // El vector tiene dos tablones más: tanto el primero como el último se consideran fuera del puente
int actual ← 0
int proximo ← 0
while actual ≤ cantidadTablones do
   proximo ← calcularProximoTablon(puente, actual, maxSalto)
   if proximo == −1 then
      return vector vacío
   end if
   introducirAlFinal(saltos, proximo)
   if proximo > cantidadTablones then
      return saltos
   end if
   actual ← proximo
end while
```

Algorithm 2 int calcularProximoTablon(vector<int> puente, int actual, int maxSalto)

```
int cantidadTablones ← |puente| − 2
while maxSalto > 0 do
   if actual + maxSalto > cantidadDeTablones then
      return cantidadDeTablones + 1
   end if
   if puente[actual + maxSalto] == 1 then
      return actual + maxSalto
   end if
   maxSalto ← maxSalto − 1
end while
return −1
```

1.3. Demostración

Vamos a tratar de probar que dado una secuencia de saltos, si para cada salto s, s es un ”salto maximoz si la sumatoria de saltos es mayor a la cantidad de tablones del puente, entonces nuestra secuencia es solucion del problema.

Dada un Sec<Salto> *se*.

$(\forall i : Nat, i < se.long)(esMax(se_i) \wedge \sum_{j=0}^{se.long-1} se_j = puente.long) \implies$
$\nexists(se' : Sec < Salto >)/se.long < se'.long \wedge \sum_{j=0}^{se'.long-1} se'_j \geq puente.long)$

Para probar esto, vamos a utilizar la definición canónica de demostración de correctitud para algoritmos greedy, dada en clase por la cátedra. Un algoritmo goloso, puede plantearse como el siguiente esquema:

$S^{opt} = \emptyset$
while S $! = \emptyset$ **do**
  X $= f(S)$
  S $= S - X$
  if p(S^{opt},X) **then**
    $S^{opt} = S^{opt} \cup X$
  end if
end while
return S^{opt}

Luego para garantizar la correctitud del mismo hay que garantizar 4 puntos:

1) Definimos una noción de lo que significa ser "sub-solución" de una solución.

2) Probar que S^{opt} empieza siendo sub-solución de alguna solución óptima.

3) Probar que si S^{opt} es sub-solución de alguna solución óptima al iniciar una iteración del ciclo, entonces al terminar esa iteración, S^{opt} sigue siendo una subsolución de alguna solución óptima.

4) Probar que cuando S $== \emptyset$, S^{opt} es una solución óptima.

Bueno para probar estos 4 puntos, primero vamos a definir F, S y P.

S es un multiconjunto de los saltos que uno puede dar representados por los pares (x,y) de tal que el representa un salto desde la posicion x hasta la posicion y con 0$<$ y-x$\leq$salto máx. F es una función selecciona de S el par con mayor diferencia (y - x) y menor coordenada x. P(S^{opt},X) devuelve TRUE cuando el tablón donde cae X no esta roto y cuando el salto no se superpone con un salto anterior. Vamos a definir ser sub-solución como ser el prefijo de una solución cuando ordenamos por orden de saltos. Como la cantidad de saltos que puede darse en un puente es finita, va existir al menos un máximo local y vamos a poder asignarle un cardinal. Entonces como existe al menos un conjunto de saltos de cardinal óptimo, el conjunto $\emptyset$ empieza siendo sub-solución de ese conjunto. Con esto queda comprobado 1) y 2).

Dado S^{opt} sub-solución al principio del ciclo, queremos garantizar que S^{opt} es sub-solución al finalizarlo. Supongamos que S^{opt} es sub-solución de S^* al principio del ciclo. Bueno si a S^{opt} no le agrego nada, sigue siendo una subsolucioón. Si el caso es en el que agrego X, quiero ver que eso sigue siendo alguna sub-solución para algún S'^*. En particular queremos ver que $S'^* \subset S^*$. Si agregamos X, es el caso en donde P dio true, por lo tanto X cae en un tablón que no esta roto y el salto no se superpone con uno anterior. Además X era el máximo de los saltos posibles y de menor coordenada x obtenido por F. Bueno, si X $\in S^*$, entonces podemos tomar $S'^* = S^*$, asi que veamos el caso en que X $\notin S^*$.

Ahora tomemos $S'^* = S^* \cup X$ Este conjunto puede tener elementos que se superponen, si no los tiene entonces S'^* es sub-solución. Si los tiene, y ordenamos por orden de salto (primera coordenada) , S'^* no puede superponerse con S^{opt} ya que S^{opt} era sub-solución de S^*. Por lo tanto si existe un elemento que se superponga, se superponen con X. Sea Y un elemento que se superpone con X. Sean Y_1, Y_2, X_1, X_2 las componentes de Y,X respectivamente. $Y_1 \geq X_2$ ya que sino X no tenia la primera componente mínima.

Si $Y_1 = X_1$ entonces $X_2 \geq Y_2$ ya que sino F hubiera agarrado primero a Y. Por lo tanto yo podria tomar $S'^* = S^* \cup X$ - Y y eso seguiria siendo sub-solución.

Si $Y_1 > X_1$ entonces hay dos casos:

Si $Y_2 \leq X_2$ entonces todo el intervalo cubierto por Y, estaria incluído en el intervalo cubierto por X, y en el intervalo I = $[X_1, Y_1]$, I $\neq \emptyset$ habría que cubrirlo con un salto por lo tanto S^* no sería subsolución.

Si $Y_2 > X_2$ entonces yo se que X,Y no pueden estar en la misma solución(se superponen). Pero se que Y pertenece a alguna solución. Veamos que existe un intervalo que cubre X, pero no Y que seria el intervalo I = $[X_1, Y_2]$. Bueno pero como X contiene a todo este intervalo, existe una solucion que contiene al salto $S_Y = (X_1, Y_2)$. Asi que dado un solución que contiene a Y, al menos contiene un salto más en ese intervalo. Luego dado el intervalo I'=$[X_2, Y_2]$, como X se superpone con Y, $X_2 > Y_1$, por lo tanto existe un salto S_X que con componentes (X_2, Y_2). Luego si analizamos las dos sub-soluciones contienen al mismo intervalo en dos saltos, por lo que si existe una solución para que contiene a Y, existe una solución que contenga a X.

Con esto queda comprobado 3).

Finalmente, si S == $\emptyset$ entonces no hay mas saltos, por lo tanto nuestra S^{opt} recorre todo el puente. Tenemos S^{opt} que sabemos es una sub-solución de S^*. Y si S^{opt} ya recorrio todo el puente, no hace falta agregarle ningún salto, por lo tanto $S^{opt} = S^*$.

Con esto queda comprobado 4).

LIMPIADA DE CARA FER

Definimos un salto s como un natural mayor a 0 y menor o igual a la distancia máxima que puede recorrer el participante, de sólo un salto, medida en tablones

$$s \in Saltos \Leftrightarrow (s \in \mathbb{N}_{>0} \land s \leq dist_{max})$$

Definimos un puente como una función $p : \mathbb{N}_{>0} \rightarrow \mathbb{N}$

$$p(i) = \begin{cases} 1 & \text{i} \leq 0 \\ 0 & \text{i} > \#tablones \\ 0 & \text{i} > 0 \land i \leq \#tablones \land i \in Tablones \\ 1 & \text{i} > 0 \land i \leq \#tablones \land i \notin Tablones \end{cases}$$

Para cada posición i del puente definimos su salto máximo s_{max} como

$$s_{max} = max\{n \in \mathbb{N}_{>0} \mid n \leq dist_{max} \land \neg p(n)\}$$

Nuestra implementacion recorre el puente dando saltos, garantizando que en cada salto, la distancia recorrida es máxima. Es decir, no existe otro salto tal que la distancia desde donde estamos parados es mayor a la del salto actual y el tablón en el que caes no esta roto.
Distancia es un Nat > 0.
Salto es Nat tal que $\forall s : Salto, s > 0 \land s \leq Distancia$

Vamos a tratar de probar que dado una secuencia de saltos, si para cada salto s, s es un "salto maximoz si la sumatoria de saltos es mayor a la cantidad de tablones del puente, entonces nuestra secuencia es solucion del problema.

Dada un Sec<Salto> *se*.

$(\forall i : Nat, i < se.long)(esMax(se_i) \wedge \sum_{j=0}^{se.long-1} se_j = puente.long) \implies$
$\nexists (se' : Sec < Salto >)/se.long < se'.long \wedge \sum_{j=0}^{se'.long-1} se'_j \geq puente.long)$ //

Llamemos a sMax a la secuencia de saltos maximos obtenida. Supongamos que existe secuencia s de saltos tal que la cantidad de elementos de s es menor a sMax y la sumatoria de saltos es igual o mayor a la cantidad de tablones. Bueno en particular, existe al menos un salto s_i, tal que s_i, es mayor a $sMax_i$, ya que si todos los s_i, son menores a su correspondiente $sMax_i$, entonces la sumatoria de sMax es mayor que la sumatoria de s. (comprobar esto ad-hoc, probablemente sale por induccion). Bueno, supongamos que agarro el primero de todos los s_i, que es mas grande que su correspondiente $sMax_i$. Hasta ese momento las dos subsecuencias (desde el principio hasta el elemento i) pesan lo mismo, entonces s_i esta parado en el mismo lugar y hace un salto mas grande que el salto maximo ($sMax_i$), lo cual es absurdo. Por lo tanto queda comprobado que ese s_i, no puede existir y la solucion es máxima.

1.4. Análisis de complejidad

El algoritmo *cruzarPuentes* se puede dividir en dos partes:

- Inicialización
- Ciclo

En la Inicialización el algoritmo asigna 3 variables en $O(1)$, con lo cual su complejidad es $3 * O(1) = O(1)$ El ciclo, como peor caso, itera hasta n veces, donde n es la *cantidadDeTablones*. Adentro del ciclo se calcula la función *calcularProximoTablon* para cada iteración. Esta nos dice el índice del próximo tablón óptimo para saltar, y a su vez es otro ciclo que se repite k veces haciendo una cantidad acotada de operaciones $O(1)$, donde k es la variable de entrada *maxsalto* que es un valor acotado. Luego el ciclo continua haciendo asignaciones y condicionales y devoluciones en $O(1)$.

Haciendo el calculo de complejidad obtenemos:

$O(cruzarPuentes) = 3 * O(1) + n(O(k))$

Que es lo mismo que:

$O(cruzarPuentes) = O(n * k)$

Podemos ver que el algoritmo depende de k, es decir, del *maxsalto* del participante, con lo cual podemos considerar que tiene una complejidad pseudopolinomial ya que depende de una variable de entrada, pero como sabemos que k es acotado por una constante finalmente podemos concluir que es de orden $O(n)$.

1.5. Test de complejidad

1.6. Testing

2. Problema 2: Horizontes lejanos

2.1. Presentación del problema

Dado un conjunto de rectángulos en un plano, todos apoyados sobre una linea recta horizontal, como en las siguientes figuras, se pide eliminar las líneas que colisionen con algún otro rectángulo, donde colisionar también es sólamente "tocar" otra línea.

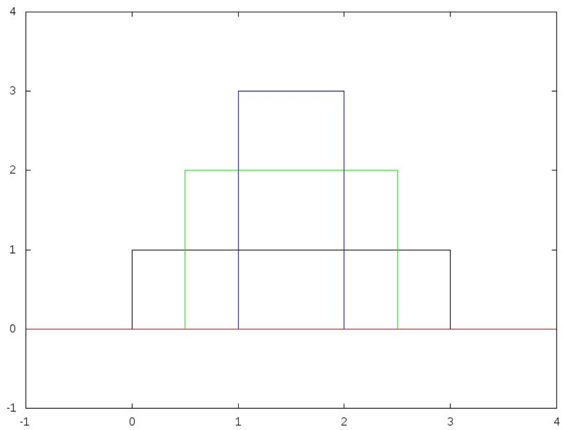

Figura 1: Con colisión total

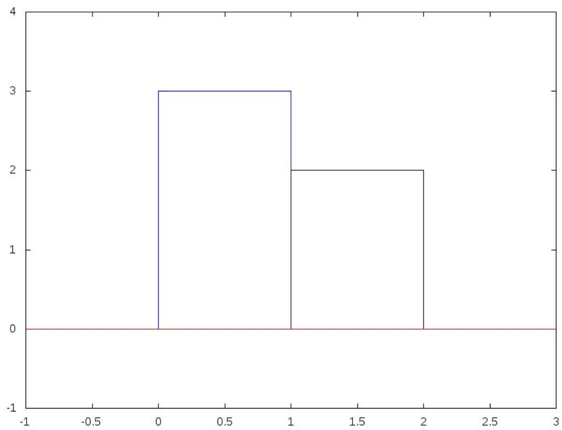

Figura 2: Sólo se tocan los bordes

Así, tras ejecutar el algoritmo, el resultado para los ejemplos anteriores sería:

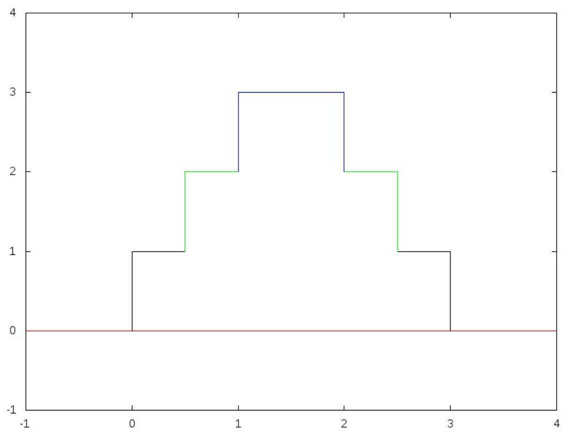

Figura 3: Resultado con colisión total

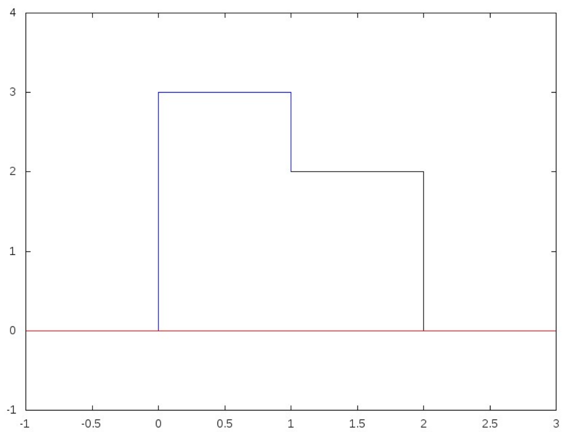

Figura 4: Resultado cuando sólo se tocan los bordes

Como requerimiento adicional, el algoritmo para n rectángulos debe tener una complejidad temporal estrictamente menor que $O(n^2)$.

2.2. Resolución

2.2.1. Algoritmo

COMPLETAR

2.2.2. Pseudocódigo

```
input: edificios
while quedan edificios do
  if empieza edificio then
    registro el edificio como abierto
    if altura del edificio es mayor a la del contorno then
      agrego la altura del edificio al contorno
    end if
  else
    saco al edificio de los abiertos
    if este edificio le daba la altura al contorno then
      agrego la altura del edificio abierto que le siga en altura al contorno
    end if
  end if
end while
return contorno
```

2.3. Demostración

2.4. Análisis de complejidad

2.5. Test de complejidad

2.6. Testing

3. Problema 3: Biohazard

3.1. Presentación del problema

3.2. Resolución

3.2.1. Algoritmo

3.2.2. Pseudocódigo

3.3. Demostración

3.4. Análisis de complejidad

3.5. Test de complejidad

3.6. Testing